# 面向对象编程



## 学习目标

- 掌握面向对象编程中类的基本定义与语法
- 学会使用类修饰符与寄存器
- 理解并掌握类的实例成员与类的静态成员的区别与使用
- 理解类与接口的关系，并熟练使用它们
- 了解类（构造函数）类型与对象类型的区别

## 类

面向对象编程中一个重要的核心就是：`类`，当我们使用面向对象的方式进行编程的时候，通常会首先去分析具体要实现的功能，把特性相似的抽象成一个一个的类，然后通过这些类实例化出来的具体对象来完成具体业务需求。

### 类的基础

在类的基础中，包含下面几个核心的知识点，也是 `TypeScript` 与 `EMCAScript2015+` 在类方面共有的一些特性

- `class` 关键字

- 构造函数：`constructor`
- 成员属性定义
- 成员方法
- this关键字

除了以上的共同特性以外，在 `TypeScript` 中还有许多 `ECMAScript` 没有的，或当前还不支持的一些特性，如：抽象

## class

通过 `class` 就可以描述和组织一个类的结构，语法：

```
// 通常类的名称我们会使用 大坨峰命名 规则，也就是 （单词）首字母大写
class User {
  // 类的特征都定义在 {} 内部
}
```

## 构造函数

通过 `class` 定义了一个类以后，我们可以通过 `new` 关键字来调用该类从而得到该类型的一个具体对象：也就是实例化。

为什么类可以像函数一样去调用呢，其实我们执行的并不是这个类，而是类中包含的一个特殊函数：构造函数 - `constructor`

```
class User {

    constructor() {
    console.log('实例化...')
  }

}
let user1 = new User;
```

- 默认情况下，构造函数是一个空函数
- 构造函数会在类被实例化的时候调用
- 我们定义的构造函数会覆盖默认构造函数
- 如果在实例化（new）一个类的时候无需传入参数，则可以省略 `()`
- 构造函数 `constructor` 不允许有 `return` 和返回值类型标注的（因为要返回实例对象）

通常情况下，我们会把一个类实例化的时候的初始化相关代码写在构造函数中，比如对类成员属性的初始化赋值

## 成员属性与方法定义

```
class User {
  id: number;
  username: string;

  constructor(id: number, username: string) {
    this.id = id;
    this.username = username;
  }

    postArticle(title: string, content: string): void {
```

```
    console.log(`发表了一篇文章： ${title}`)
  }
}

let user1 = new User(1, 'zMouse');
let user2 = new User(2, 'MT');
```

## this 关键字

在类内部，我们可以通过 `this` 关键字来访问类的成员属性和方法

```
class User {
  id: number;
  username: string;

    postArticle(title: string, content: string): void {
    // 在类的内部可以通过 `this` 来访问成员属性和方法
    console.log(`${this.username} 发表了一篇文章： ${title}`)
  }
}
```

## 构造函数参数属性

因为在构造函数中对类成员属性进行传参赋值初始化是一个比较常见的场景，所以 `ts` 提供了一个简化操作：给构造函数参数添加修饰符来直接生成成员属性

- `public` 就是类的默认修饰符，表示该成员可以在任何地方进行读写操作

```
class User {

  constructor(
    public id: number,
    public username: string
  ) {
    // 可以省略初始化赋值
  }

    postArticle(title: string, content: string): void {
    console.log(`${this.username} 发表了一篇文章： ${title}`)
  }
}

let user1 = new User(1, 'zMouse');
let user2 = new User(2, 'MT');
```

## 继承

在 `ts` 中，也是通过 `extends` 关键字来实现类的继承

```
class VIP extends User {

}
```

## super 关键字

在子类中，我们可以通过 `super` 来引用父类

- 如果子类没有重写构造函数，则会在默认的 `constructor` 中调用 `super()`
- 如果子类有自己的构造函数，则需要在子类构造函数中显示的调用父类构造函数：`super(//参数)`，否则会报错
- 在子类构造函数中只有在 `super(//参数)` 之后才能访问 `this`
- 在子类中，可以通过 `super` 来访问父类的成员属性和方法
- 通过 `super` 访问父类的的同时，会自动绑定上下文对象为当前子类 `this`

```
class VIP extends User {

  constructor(
        id: number,
      username: string,
      public score = 0
    ) {
        super(id, username);
    }

  postAttachment(file: string): void {
    console.log(`${this.username} 上传了一个附件： ${file}`)
  }
}

let vip1 = new VIP(1, 'Leo');
vip1.postArticle('标题', '内容');
vip1.postAttachment('1.png');
```

### 方法的重写与重载

默认情况下，子类成员方法集成自父类，但是子类也可以对它们进行重写和重载

```
class VIP extends User {

    constructor(
        id: number,
      username: string,
      public score = 0
    ) {
        super(id, username);
    }

    // postArticle 方法重写，覆盖
    postArticle(title: string, content: string): void {
      this.score++;
      console.log(`${this.username} 发表了一篇文章： ${title}，积分：
${this.score}`);
    }

    postAttachment(file: string): void {
        console.log(`${this.username} 上传了一个附件： ${file}`)
    }
}
```

```
// 具体使用场景
let vip1 = new VIP(1, 'Leo');
vip1.postArticle('标题', '内容');
```

```
class VIP extends User {

    constructor(
        id: number,
      username: string,
      public score = 0
    ) {
        super(id, username);
    }

    // 参数个数，参数类型不同：重载
    postArticle(title: string, content: string): void;
    postArticle(title: string, content: string, file: string): void;
    postArticle(title: string, content: string, file?: string) {
        super.postArticle(title, content);

        if (file) {
            this.postAttachment(file);
        }
    }

    postAttachment(file: string): void {
        console.log(`${this.username} 上传了一个附件： ${file}`)
    }
}

// 具体使用场景
let vip1 = new VIP(1, 'Leo');
vip1.postArticle('标题', '内容');
vip1.postArticle('标题', '内容', '1.png');
```

# 修饰符

有的时候，我们希望对类成员（属性、方法）进行一定的访问控制，来保证数据的安全，通过 `类修饰符` 可以做到这一点，目前 TypeScript 提供了四种修饰符：

- public：公有，默认
- protected：受保护
- private：私有
- readonly：只读

## public 修饰符

这个是类成员的默认修饰符，它的访问级别为：

- 自身
- 子类
- 类外

## protected 修饰符

它的访问级别为：

- 自身
- 子类

## private 修饰符

它的访问级别为：

- 自身

## readonly 修饰符

只读修饰符只能针对成员属性使用，且必须在声明时或构造函数里被初始化，它的访问级别为：

- 自身
- 子类
- 类外

```
class User {

  constructor(
    // 可以访问，但是一旦确定不能修改
    readonly id: number,
    // 可以访问，但是不能外部修改
    protected username: string,
    // 外部包括子类不能访问，也不可修改
    private password: string
  ) {
    // ...
  }
    // ...
}

let user1 = new User(1, 'zMouse', '123456');
```

# 寄存器

有的时候，我们需要对类成员 `属性` 进行更加细腻的控制，就可以使用 `寄存器` 来完成这个需求，通过 `寄存器`，我们可以对类成员属性的访问进行拦截并加以控制，更好的控制成员属性的设置和访问边界，寄存器分为两种：

- getter
- setter

## getter

访问控制器，当访问指定成员属性时调用

## setter- 组件

- 函数式组件

- 类式组件

- props 与 state

- 组件通信

- 表单与受控组件

设置控制器，当设置指定成员属性时调用

```
class User {

    constructor(
        readonly _id: number,
      readonly _username: string,
      private _password: string
    ) {
    }

    public set password(password: string) {
        if (password.length >= 6) {
            this._password = password;
        }
    }

    public get password() {
        return '******';
    }
    // ...
}
```

## 静态成员

前面我们说到的是成员属性和方法都是实例对象的，但是有的时候，我们需要给类本身添加成员，区分某成员是静态还是实例的：

- 该成员属性或方法是类型的特征还是实例化对象的特征
- 如果一个成员方法中没有使用或依赖 `this` ，那么该方法就是静态的

```
type IAllowFileTypeList = 'png'|'gif'|'jpg'|'jpeg'|'webp';

class VIP extends User {

  // static 必须在 readonly 之前
  static readonly ALLOW_FILE_TYPE_LIST: Array<IAllowFileTypeList> =
['png','gif','jpg','jpeg','webp'];

  constructor(
        id: number,
      username: string,
      private _allowFileTypes: Array<IAllowFileTypeList>
    ) {
        super(id, username);
  }

  info(): void {
    // 类的静态成员都是使用 类名.静态成员 来访问
```

```
    // VIP 这种类型的用户允许上传的所有类型有哪一些
    console.log(VIP.ALLOW_FILE_TYPE_LIST);
    // 当前这个 vip 用户允许上传类型有哪一些
    console.log(this._allowFileTypes);
  }
}

let vip1 = new VIP(1, 'zMouse', ['jpg','jpeg']);
// 类的静态成员都是使用 类名.静态成员 来访问
console.log(VIP.ALLOW_FILE_TYPE_LIST);
this.info();
```

- 类的静态成员是属于类的，所以不能通过实例对象（包括 this）来进行访问，而是直接通过类名访问（不管是类内还是类外）
- 静态成员也可以通过访问修饰符进行修饰
- 静态成员属性一般约定（非规定）全大写

# 抽象类

有的时候，一个基类（父类）的一些方法无法确定具体的行为，而是由继承的子类去实现，看下面的例子:

现在前端比较流行组件化设计，比如 `React`

```
class MyComponent extends Component {

  constructor(props) {
    super(props);
    this.state = {}
  }

  render() {
    //...
  }

}
```

根据上面代码，我们可以大致设计如下类结构

- 每个组件都一个 `props` 属性，可以通过构造函数进行初始化，由父级定义
- 每个组件都一个 `state` 属性，由父级定义
- 每个组件都必须有一个 `render` 的方法

```
class Component<T1, T2> {

    public state: T2;

    constructor(
        public props: T1
    ) {
        // ...
    }

    render(): string {
```

```
        // ...不知道做点啥才好，但是为了避免子类没有 render 方法而导致组件解析错误，父类就用一个默认的 render 去处理可能会出现的错误
    }
}

interface IMyComponentProps {
    title: string;
}
interface IMyComponentState {
    val: number;
}
class MyComponent extends Component<IMyComponentProps, IMyComponentState> {

    constructor(props: IMyComponentProps) {
        super(props);

        this.state = {
            val: 1
        }
    }

    render() {
        this.props.title;
        this.state.val;
        return `<div>组件</div>`;
    }

}
```

上面的代码虽然从功能上讲没什么太大问题，但是我们可以看到，父类的 `render` 有点尴尬，其实我们更应该从代码层面上去约束子类必须得有 `render` 方法，否则编码就不能通过

## abstract 关键字

如果一个方法没有具体的实现方法，则可以通过 abstract 关键字进行修饰

```
abstract class Component<T1, T2> {

    public state: T2;

    constructor(
        public props: T1
    ) {
    }

    public abstract render(): string;
}
```

使用抽象类有一个好处：

约定了所有继承子类的所必须实现的方法，使类的设计更加的规范

使用注意事项：

> - abstract 修饰的方法不能有方法体
> - 如果一个类有抽象方法，那么该类也必须为抽象的

> - 如果一个类是抽象的，那么就不能使用 new 进行实例化（因为抽象类表名该类有未实现的方法，所以不允许实例化）
> - 如果一个子类继承了一个抽象类，那么该子类就必须实现抽象类中的所有抽象方法，否则该类还得声明为抽象的

# 类与接口

在前面我们已经学习了接口的使用，通过接口，我们可以为对象定义一种结构和契约。我们还可以把接口与类进行结合，通过接口，让类去强制符合某种契约，从某个方面来说，当一个抽象类中只有抽象的时候，它就与接口没有太大区别了，这个时候，我们更推荐通过接口的方式来定义契约

- 抽象类编译后还是会产生实体代码，而接口不会
- `TypeScript` 只支持单继承，即一个子类只能有一个父类，但是一个类可以实现过个接口
- 接口不能有实现，抽象类可以

## implements

在一个类中使用接口并不是使用 `extends` 关键字，而是 `implements`

- 与接口类似，如果一个类 `implements` 了一个接口，那么就必须实现该接口中定义的契约
- 多个接口使用 `,` 分隔
- `implements` 与 `extends` 可同时存在

```
interface ILog {
  getInfo(): string;
}

class MyComponent extends Component<IMyComponentProps, IMyComponentState>
implements ILog {

    constructor(props: IMyComponentProps) {
        super(props);

        this.state = {
            val: 1
        }
    }

    render() {
        this.props.title;
        this.state.val;
        return `<div>组件</div>`;
    }

    getInfo() {
        return `组件：MyComponent，props：${this.props}，state：${this.state}`;
    }

}
```

实现多个接口

```
interface ILog {
    getInfo(): string;
```

```
}
interface IStorage {
    save(data: string): void;
}

class MyComponent extends Component<IMyComponentProps, IMyComponentState>
implements ILog, IStorage {

    constructor(props: IMyComponentProps) {
        super(props);

        this.state = {
            val: 1
        }
    }

    render() {
        this.props.title;
        this.state.val;
        return `<div>组件</div>`;
    }

    getInfo(): string {
        return `组件：MyComponent，props：${this.props}，state：${this.state}`;
    }

    save(data: string) {
        // ... 存储
    }

}
```

接口也可以继承

```
interface ILog {
    getInfo(): string;
}
interface IStorage extends ILog {
    save(data: string): void;
}
```

# 类与对象类型

当我们在 TypeScript 定义一个类的时候，其实同时定义了两个不同的类型

- 类类型（构造函数类型）
- 对象类型

首先，对象类型好理解，就是我们的 new 出来的实例类型

那类类型是什么，我们知道 JavaScript 中的类，或者说是 TypeScript 中的类其实本质上还是一个函数，当然我们也称为构造函数，那么这个类或者构造函数本身也是有类型的，那么这个类型就是类的类型

```
class Person {
    // 属于类的
  static type = '人';

  // 属于实例的
  name: string;
  age: number;
  gender: string;

  // 类的构造函数也是属于类的
  constructor( name: string, age: number, gender: '男'|'女' = '男' ) {
    this.name = name;
    this.age = age;
    this.gender = gender;
  }

  public eat(): void {
    // ...
  }

}

let p1 = new Person('zMouse', 35, '男');
p1.eat();
Person.type;
```

上面例子中，有两个不同的数据

- `Person` 类（构造函数）
- 通过 `Person` 实例化出来的对象 `p1`

对应的也有两种不同的类型

- 实例的类型（`Person`）
- 构造函数的类型（`typeof Person`）

用接口的方式描述如下

```
interface Person {
    name: string;
    age: number;
    gender: string;
    eat(): void;
}

interface PersonConstructor {
    // new 表示它是一个构造函数
    new (name: string, age: number, gender: '男'|'女'): PersonInstance;
    type: string;
}
```

在使用的时候要格外注意

```
function fn1(arg: Person /*如果希望这里传入的Person 的实例对象*/) {
    arg.eat();
}
fn1( new Person('', 1, '男') );

function fn2(arg: typeof Person /*如果希望传入的Person构造函数*/) {
    new arg('', 1, '男');
}
fn2(Person);
```